Testing

Lecture 15
Definition of Testing

- What is “testing”?  
  - A process whereby we increase our confidence in an implementation by observing its behavior

- Fundamental point:  
  - Testing can detect the presence of mistakes, never their absence!

- Fail a test case ==>  
- Pass all test cases ==>
Importance of Testing

- Despite limitations, testing is the most practical approach for large systems
- Donald Knuth quotation:
  "Warning: I’ve only proven this algorithm is correct... I haven’t tested it!"
Theory

- 3 levels of abstraction in functionality
- Want: the idea
- Have: an implementation
- “Testing” requires comparing it against something, but what?
Theory (II)

- Ideal: test against our “idea”
  - But the idea is usually too fuzzy
- So make it concrete by writing a specification
  - Defines desired mapping from input to output

```
Input → Specification → Expected Output
            /              /
         /                /
Implementation     Actual Output
```
Example: Sorting a List

- Idea: Function takes a list and puts it in order
- Too fuzzy!
- Questions:
  - Does it modify the list or return a new one?
  - Does it require the list to be non empty?
  - Does it sort in increasing or decreasing order?
  - What kind of items can be in the list?
Example: Sorting a List

- Specification: Describe how inputs map to outputs
- Recall software I/II contracts

  requires
  \[ |list| \leq 65535 \]

  ensures
  \[ \forall i: 0 \leq i < list.length: list[i] \leq list[i + 1] \]
Example: Sorting a List

- Specification: Describe how inputs map to outputs
- Recall software I/II contracts
  requires
  \[|\text{list}| \leq 65535\]
  ensures
  \[\forall i: 0 \leq i < \text{list}.\text{length} - 1: \text{list}[i] \leq \text{list}[i + 1]\]
Example: Sorting a List

- Specification: Describe how inputs map to outputs
- Recall software I/II contracts
  requires
    \[ |list| \leq 65535 \]
  ensures
    \[ \forall i : 0 \leq i < list.\ length - 1 : list[i] \leq list[i + 1] \]
    \[ list = permutation(\#list) \]
Initial and Final Values

- Often the final value of a parameter depends on its initial value:
  - SW I/II: "Updates" parameter mode
  - Example: mutator method sort!

- Consequence: Specification includes old value (e.g., #list) in ensures clause.

- Sometimes the final value is independent of its initial value:
  - "Replaces" and "Clears" parameter modes
  - Example: mutator methods fill!, clear!
Relational Specifications

- A function maps each element in its domain to a *single* element in its range.
- A relation maps each element in its domain to *at least* one elt in its range.
- For specifications:
  - Function = deterministic behavior
  - Relation = nondeterministic behavior
- Examples
  - `find_factor n` returns *some* prime factor
  - `shuffle` scrambles elements of array
Frame of Mind for Validation

- Tests should be written to *break* a program
  - Not to show it works!
- When a test reveals an error, that’s *success*!
  - Failed test case is a *positive* thing
- Good approach: have *someone else* test your code
Importance of Indep't Testing

- See IEEE Computer, Oct ‘99
  - study at NASA Langley
  - had two groups working in parallel
- The group with independent testers found:
  - more faults overall (critical and non-critical)
  - found these faults earlier in the process
  - fixed these faults with less effort
Figure 1 from Arthur article

The diagram shows a comparison between the IV&V group and the Non-IV&V group in terms of the number of critical faults at different stages of the development process. The stages are Requirements, HLD, LLD, Code/UT, I&T, and Total.

- Requirements: IV&V group has 16 faults, Non-IV&V group has 0 faults.
- HLD: IV&V group has 20 faults, Non-IV&V group has 2 faults.
- LLD: IV&V group has 31 faults, Non-IV&V group has 8 faults.
- Code/UT: IV&V group has 24 faults, Non-IV&V group has 34 faults.
- I&T: IV&V group has 6 faults, Non-IV&V group has 14 faults.
- Total: IV&V group has 97 faults, Non-IV&V group has 58 faults.
Figure 2 from Arthur article

The graph shows the effort in minutes for different categories of faults for two groups: IV&V group and Non-IV&V group. The categories are Noncritical, Critical, and Combined faults.

- **Noncritical**:
  - IV&V group: 4 minutes
  - Non-IV&V group: 7 minutes

- **Critical**:
  - IV&V group: 21 minutes
  - Non-IV&V group: 11 minutes

- **Combined**:
  - IV&V group: 7 minutes
  - Non-IV&V group: 20 minutes
Writing Good Tests (Inputs)

- Too many possible inputs to test them all
  - Space is defined by requires clause
  - Choose inputs wisely
- Test boundary conditions
  - eg 0, empty array, empty string
- Test different categories of input
  - eg positive, negative, and zero
- Test different categories of behavior
  - eg each menu option, each error message
- Test “unexpected” input
  - eg nil, last name includes a space
- Test representative “normal” input
  - eg random, reasonable values
How To Create Expected Output

- By hand
  - Error-prone and tedious
- With another program
  - Also error-prone
  - Often just redoing the implementation, and making the same mistakes!
- Work backwards
  - Inverse may be easier to calculate
  - Eg start with a sorted list then permute it
Alternative: Validating Output

- Steps:
  - Keep a copy of the input
  - Run the program
  - Validate the actual output against input

- Example: sorting
  - write two helper functions:
    1. copy the input
    2. run program and check

- Helper functions may be easier to get right than the unit under test
Dangers with Testing

- “Expected output” is wrong
- Testing program is wrong
  - Extra code means more chances to make mistakes
  - E.g. `permute(a,b)` always returns true
- With these errors, there are 2 dangers:
  1. Spurious test failures (passes when shouldn't)
  2. False positives (fails when it shouldn't)
- Which is worse?
Another Danger with Testing

- A third, more subtle, potential error: The specification is wrong
- How can this be?

- When testing drives implementation, this kind of error will not be exposed
- To increase the chances of finding these problems, have someone else test your code!
Levels of Testing

Different kinds of testing, aimed at identifying different kinds of errors

1. Unit tests
2. Integration tests
3. System tests
Unit Tests

- Individual components tested in isolation
  - UUT: Unit Under Test
- Often uses a test fixture
  - Configuration, values, objects which are set up before running all the tests
- Flavors of unit testing:
  - *Black box*: testing based *only* on specification (tester does not look at code)
  - *White box*: testing based on code structure (tester looks at code to make sure every branch of a switch statement is followed)
Integration Tests

- Modules tested in combination in order to check the *interfaces*
- Best done incrementally

Diagram:
- Driver
  - Initialize
  - Load
    - FileIO
    - Header
    - Reserve
  - Simulate
- SetMemory
Bottom-up vs Top-down Testing

**Bottom-up**
- Start with most basic modules
- Easy to exercise all the features
- Write a “driver” for higher-level modules

**Top-down**
- Start at top (main)
- Test interfaces early
- Write “stubs” for lower level modules

Often these two occur simultaneously, in tandem
System Tests

- Verify that system as a whole meets the requirements and specifications

- Three flavors:
  - alpha: By developers, before release
  - beta: By "friendly customers", before general release
  - acceptance: By end customer, to decide whether or not to hire you next time!
TDD: Test-Driven Development

- In dynamic languages, testing is even more important
  - Load-time errors << compile-time errors
- Extreme position: "If it isn't tested, assume it doesn't work"
- TDD: Write tests first (before code)
  - Recall "client-view first" in Software I/II
  - Development cycle: "red, green, refactor"
    - Write tests, watch them fail
    - Write (only) enough code for tests to pass (may need to refactor)
    - Repeat
Summary

- Nature of testing
  - Specification, implementation, test cases
  - Initial values matter too
- Importance of the right frame of mind
  - Write tests to break code
  - TDD: write tests to guide development
- Levels of Testing
  - Unit tests
  - Integration tests
  - System tests
Testing Frameworks (MiniTest)
MiniTest and RSpec

- Many popular testing libraries for Ruby
- MiniTest (replaces older Test::Unit)
  - Comes built-in
  - Looks like JUnit (mapped to Ruby syntax)
  - Well-named!
- RSpec
  - Installed as a library (ie a "gem")
  - Looks different from JUnit (and even Ruby!)
  - Most unfortunate name!
- RSpec view is that test cases *define* expected behavior... ie they *are* the spec!
Writing MiniTest Tests

- Require runner and UUT
  ```ruby
  require 'minitest/autorun'
  require 'card'
  ```

- Test fixture = subclass of Minitest::Test
  ```ruby
class TestCard < Minitest::Test
  ```

- Test case = method in the fixture
  - Method name must begin with `test_`
  ```ruby
def test_identifies_set ... end
  ```
  - Contains assertion(s) exercising a single piece of code / behavior / functionality
  - Should be small (ie test one thing)
  - Should be independent (ie of other tests)

- Test Suite = collection of fixtures
Example: test_card.rb

```ruby
require 'minitest/autorun'
require 'card'  # assume card.rb on load path

class TestCard < Minitest::Test
  
  def test_has_number
    assert_respond_to Card.new, :number
  end

  def test_remembers_number
    @card = Card.new 1, "oval", "open", "red"
    assert_equal 1, @card.number
  end

end
```
Execution Model

TestCard

instance of

@card

instance of

has_number()
remembers()
Execution Model: Implications

- Separate instances of test class created
  - One instance / test case
- Test cases don't have side effects
  - Passing/failing one test does not affect others
- Can not rely on order of tests
  - Randomized order of execution
  - Controllable with --seed command-line option
  - Also controllable by invoking
    `i_suck_and_my_tests_are_order_dependent!`
- Fixture: common set-up to all test cases
  - Field(s) for instance(s) of class being tested
  - Factor initialization code into its own method
  - This method must be called `setup`
Good Practice: setup

- Initialize a fixture with a setup method (rather than initialize method)

- Reasons:
  - If the code being tested throws an exception *during the setup*, the output is much more meaningful
  - Symmetry with teardown method for cleaning up after a test case
Example: test_card.rb

```ruby
require 'minitest/autorun'
require 'card' #assume card.rb is on load path

class TestCard < Minitest::Test

  def setup
    @card = Card.new 1, "oval", "open", "red"
  end

  def test_has_number
    assert.respond_to @card, :number
  end

  def test_remembers_number
    assert_equal 1, @card.number
  end
end
```
Execution Model

TestCard

instance of

@card
setup()
has_number()
remembers()

instance of

@card
setup()
has_number()
remembers()
MiniTest Assertion Methods

- Most have two versions: assert & refute
  - Example: `assert_nil, refute_nil`
  - No need for negation (use refute instead)
- Most take an optional message
  ```ruby
  assert_empty Library.new, "A new library contains no books"
  ```
  - Message appears when assertion fails
- Specials:
  - `pass/flunk` – always passes/fails
  - `skip` – skips the rest of the test case
- Performance benchmarking also available
Common Assertions

- **Boolean condition:** `assert/refute`
  ```ruby
  assert @books.all {|b| b.available?}
  ```

- **Is nil:** `(assert/refute) nil`
  ```ruby
  refute_nil @library.manager
  ```

- **Is empty:** `(assert/refute) empty`
  ```ruby
  assert_empty Library.new
  ```
Asserting Equality

- Two objects are `==` equal
  
  ```ruby
  assert_equal expected, actual
  ```
  
  - Compares `object` values (ie `==` in Ruby)
  - Failure produces useful output
    
    ```ruby
    TestCard#test_total_number_of_cards
    Expected: 81
    Actual: 27
    ```
    
    - Compare with basic `assert exp == actual`
      
      ```ruby
      TestCard#test_shuffle_is_permutation
      Failed assertion, no message given
      ```

- Two objects are the same (aliases)
  
  ```ruby
  assert_same @table.north, @players.first
  ```
  
  - Compares `reference` values (ie `.equal?`)

- Two objects are the same (aliases)
  
  ```ruby
  assert_same @table.north, @players.first
  ```
  
  - Compares `reference` values (ie `.equal?`)

- Two objects are the same (aliases)
  
  ```ruby
  assert_same @table.north, @players.first
  ```
  
  - Compares `reference` values (ie `.equal?`)

- Two objects are the same (aliases)
  
  ```ruby
  assert_same @table.north, @players.first
  ```
  
  - Compares `reference` values (ie `.equal?`)
More Assertions

- String matches a regular expression
  ```ruby
  assert_match /CSE.*/, @course.name
  ```

- Collection includes a particular item
  ```ruby
  assert_includes @library, @book
  ```

- Object is of a particular type
  ```ruby
  assert_instance_of String, @book.title
  ```

- Object has a method
  ```ruby
  assert_respond_to @student, :alarm
  ```

- Block raises an exception
  ```ruby
  assert.raises ZeroDivisionError do
    @library.average_book_cost
  end
  ```
Good Practice: Comparing Floats

- Never compare floating point numbers directly for equality
  
  ```
  assert_equal 1.456, calculated,
  "Low-density experiment"
  ```

- Numeric instabilities make exact equality problematic for floats

- Better approach: Equality with tolerance (delta or epsilon)
  
  ```
  assert_in_delta Math::PI, (22.0 / 7.0), 0.01, "Archimedes algorithm"
  ```

- Delta for error, epsilon for relative error
Good Practice: Organization

- Keep tests in the same *project* as the code
  - They are part of the build, the repo, etc
  - Helps to keep tests current
- Separate tests and implementation
  - `\set\lib` – contains card.rb...
  - `\set\tests` – contains test_card.rb
- Name test classes consistently
  - eg TestCard tests Card
- Test fixture is a ruby program
  - `set $ ruby tests\test_card.rb`
  - Test needs to be able to find UUT (require)
  - Add location of UUT to load path
    - `set $ ruby -I lib tests\test_card.rb`
Alternative Syntax

- Problem: Cumbersome method names
  ```ruby
test_shuffle_changes_deck_configuration
  ```
- Solution: exploit Ruby language flexibility in API of testing library
  - Methods are available that change the syntax and structure of test cases
  - "Domain-specific language" for testing
- Result: MiniTest::Spec
  - Notation inspired by Rspec
Writing MiniTest::Spec Tests

- **Require spec library (+ runner + UUT)**
  ```ruby
  require 'minitest/spec'
  ```

- **Test fixture (an “example group”) is a `describe` block**
  ```ruby
  describe Card do ...
  end
  ```
  - Can be nested, and identified by string
  - The block contains *examples*

- **Test case (an “example”) is an `it` block**
  ```ruby
  it 'identifies a set' ...
  end
  ```
  - Contains *expectation(s)* on a single piece of code / behavior / functionality

- **Expectations are methods on objects**
  ```ruby
  @card.number.must_equal 1
  ```
Example: test_card.rb

```ruby
require 'minitest/spec'
require 'minitest/autorunner'
require 'card'  #assume card.rb is on load path

describe Card, 'card for game of set' do

  it 'has a number' do
    Card.new.must.respond_to :number
  end

  it 'remembers its original number' do
    @card = Card.new 1, "oval", "open", "red"
    @card.number.must.equal 1
  end

end
end```

Expectations vs. Assertions

- Positive and negative form
  - `must_be_empty`
  - `wont_be_empty`

- Have corresponding assertions
  - Different argument order
    - `assert_equal expected, actual`
    - `actual.must_equal expected`

- No string argument
  - Meaningful output comes from group and example name(s)
Obj.Must_ + ...

- be
  - x.must_be :<, 10
- equal, be_same_as
  - x.must_equal y
- be_nil
- be_empty, include
- be_instance_of, be_kind_of
- be_within_delta, be_within_epsilon
- match
  - msg.must_match /^Dear/
- respond_to
- raise
- output, be_silent
Setup/Teardown

- Methods before, after
- Arguments :each or :all

```ruby
describe Student do
  before :each do
    @buck_id = BuckID.new "4328429"
    @s = Student.new buck_id
  end

  it 'should come to class' do ... end
end
```
Let: Lazy Initialization

describe Student do
  let(:student) {Student.new 1234}

  describe 'sleep deprivation'
    it 'misses class' do
      student.awake?.must_equal false
    end
  end
end
Setting up and Using RSpec

- Install the rspec gem locally
  
  ```bash
  [~] $ gem install rspec
  ```

- Set up your program to use rspec
  
  ```bash
  [myapp] $ rspec --init
  ```

- Init creates several things in myapp/
  ```
  spec/  # put tests (foo_spec.rb) here
  spec/spec_helper.rb # configures paths
  .rspec # default command-line args
  ```

- Run tests
  
  ```bash
  [myapp] $ rspec spec/foo_spec.rb
  ```
Example Groups and Examples

```ruby
require_relative './student'

describe Student do  # example group
  it 'can drop a class' do  # example
    ...
  end

  context 'when attending lecture' do
    it 'stays awake during lecture' do
      ...
    end

    it 'stores info until exam' do
      ...
    end
  end
end
end
```
RSpec Expectations

- Verb is "should" (or "should_not")
  \texttt{target.should \textit{condition} \#notice space}
- Examples of condition
  - \texttt{==}, equal,
    \texttt{factor.should equal 34}
  - be_true, be_false, be_nil, be_empty
    \texttt{list.empty?.should be_true}
  - have(n).items, have_at_most(n).items
  - include(item)
    \texttt{list.should include(name)}
  - match(regex)
  - respond_to(method_name)
- New form: expect().to (or not_to)
  \texttt{expect(a_result).to \texttt{eq} "OSU"}
Stubs

- Top-down: testing a class that uses A, B, C
- We don't have A, B, C, we want quick approximations of A, B, C
  - Behave in certain way, returning canned answers
- Stub method
  - Takes a hash \{ method_names: return_values \}
  - returns an object with those methods
    
    ```ruby
    stub_printer = stub :available? => true,
      :render => nil
    ```

- Another form adds (or changes) a method/return value to an existing object
  ```ruby
  long_str = 'something'
  long_str.stub! (:length).and_return(1000000)
  ```
Mocks

- Stubs passively allow the test to go through
-Mocks *monitor* how they are used (and will fail if they aren't used right)

```ruby
it 'should know how to print itself' do
  mock_printer = mock('Printer')
  mock_printer.should_receive(:available?).and_return(true)
  mock_printer.should_receive(:render).exactly(3).times
  @doc.print (mock_printer).should == 'Done'
end
```
Summary

- **MiniTest**
  - Test case: method named `test_`
  - Test fixture: class extending `Minitest::Test`
- **Execution model: multiple instances**
  - Independence of test cases
- **MiniTest::Spec**
  - Examples and expectations
  - String descriptions
- **RSpec**
  - Stubs and mocks